ENG364: Microcomputer Interfacing

Laboratory Manual

School of Engineering,University of Guelph
Fall 2012

Embedded computer systems, which are electronic systems that include a microcomputer to perform a specific dedicated application, are ubiquitous. Our global economy, food production, transportation system, military defense, communication systems, and even quality of life depend on the efficiency and effectiveness of these embedded systems. As system engineers we play a major role in all phases of this effort: planning, design, analysis, manufacturing, and marketing.

This document is orgainzed as follows: Section 1 introduces the main objectives, practices and recommendations for the ENG3640 laboratory. Section 2 introduces the M68HC12 microcontroller and its different componenets. Section 3 introduces the different lab modules that will be performed by students. Section 4 gives some guidance on how to writeup the lab report. Finally, several appendices can be found at the end of this document which are useful as a reference.

1 Laboratory Objectives and Practices

The ENG364 Microcomputer Interfacing labs are an integral part of the course. The objectives of the laboratory are:

- to give you practical experience with the process of design and interfacing using Motorola MC68HC12.
- to help you understand and assimilate the lecture material.
- to give you practical experience with the process of design and implementation of analog/digital circuits.
- to give you hands-on-experience with microcontrollers such as the MC68HC12.

Laboratory Practices

1. Absolutely no food or drink in the laboratories.
2. Clean up after yourselves. Put paper in the recycling bin and garbage in the trash.
3. Do not leave the door or windows open.

The room will be closed after-hours if these rules cannot be followed.!

Laboratory Recommendations

- Labs are to be done in groups of at least 3 and at most 4 people.
- The labs are to be demonstrated during the lab period on the due date.
- All written lab reports are due during demonstration of the demo lab.

- Students are encouraged to work ahead.
- **Utilize the code available in your text book!**

2 The MC68HC11/MC68HC12 Microcontrollers

At the core of the 68HC12 is CPU12, a high-speed 16-bit evolution of the 68HC11 architecture that is designed to maintain complete source-code compatibility with the 68HC11 core. The 68HC12 fully supports all internal registers, instructions, addressing modes, and operating modes of the 68HC11. Figure 1 shows the main modules embedded in the MCU12 Microcontroller.

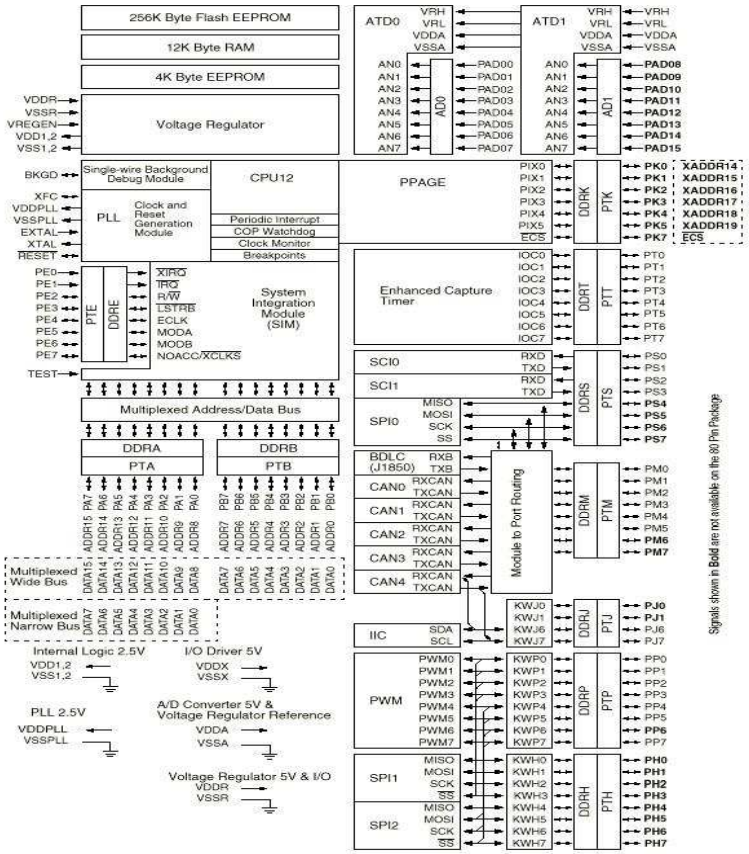

Figure 1: The MCU12 Block Diagram and Modules

Additional features and benefits include:

- Up to 8 MHz bus speed at 5 V
- Up to 5 MHz bus speed at 3 V
- 64 new instructions, 20-bit Arithmetic Logic Unit (ALU), instruction queue, and 7 new indexed addressing modes
- 16-bit data paths
- 64 Kbytes of linear and paged memory addressing capability, enabling access to more than 4 Mbytes of program space and 1 Mbyte of data space

- Call and Return from Call (RTC) instructions for efficient paged addressing
- High-level Language (HLL) optimization
- Fast math capabilities (16 x 16 MUL, 32 w 16 DIV, EMACS)
- Fuzzy logic instructions for simplified programming, reduced code size, and faster code execution
- Low-power wait and stop modes

High Performance Timer

The 68HC12 timer provides flexibility, performance, and ease of use. The system is based on a free-running, 16-bit counter with a programmable prescaler, overflow interrupt, and separate function interrupts.

Additional M68HC12 timer features include:

- Multiple timer channels
  - Each channel configurable for either input capture or output compare functions
- Real-time periodic interrupts
- Computer Operating Properly (COP) watchdog protection against software failures
- Pulse accumulator for external event counting or gated time accumulation
- An optional PWM offering up to four channels and up to 16-bit PWM outputs
- Optional event counter system for advanced timing operations

Pulse-Width Modulation

The 68HC12 Family offers a selection of Pulse-Width Modulation (PWM) options to support a variety of applications. Up to four PWM channels can be selected to create continuous waveforms with programmable rates and software selectable duty cycles from 0 to 100%.

Analog-to-Digital Converter (ADC)

The ADC periodically samples external analog signals and produces corresponding digital values. Typical applications are measuring analog inputs like battery voltage, temperature, pressure, and fluid levels. Linear successive approximation 8-bit or 10-bit resolution Single or continuous conversion modes Multiple result registers Selectable ADC clock Analog multiplexor allows variable number of channels with a single ADC.

Evaluation Board

The Dragon12-Plus2 Evaluation Board (EVB) is an economical tool for designing and debugging code for, and evaluating the operation of, the MC9S12DG256CVPE microcontroller. By providing the essential MCU timing and I/O circuitry, the EVB simplifies user evaluation of prototype hardware and software. The EVB features a prototype area, which allows custom interfacing with the microcontroller's I/O and bus lines. The Dragon12-Plus2 evaluatin board can also accommodate various types and configurations of external memory to suit a particular application's requirements.

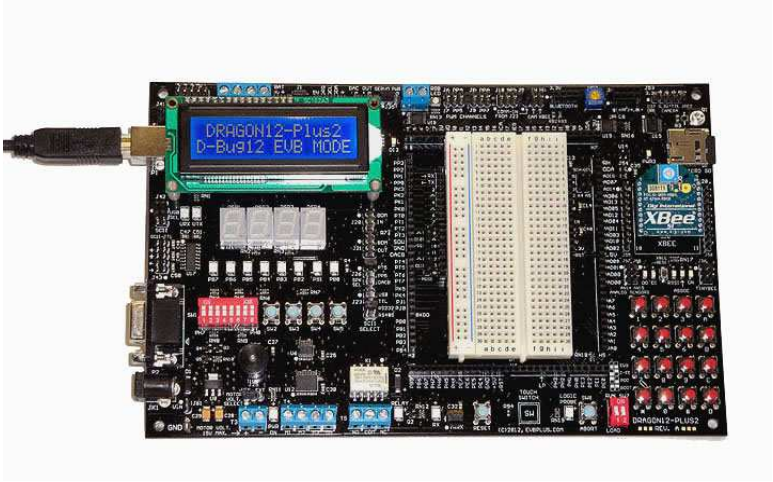

Figure 2: The Dragon12-Plus2 Evaluation Board

3 Laboratory Assignments

The course labs are structured as seven assignments. They cover the following material:

- lab0: Introduction to lab equipment and M68HC12.
- lab1: Introduction to Assembly Language Programming.
- lab2: Parallel Port Programming.
- lab3: Keypad and Switch Debouncing.
- lab4: Servo motor control and LCD interfacing.
- lab5: A/D converter and High-Tech Etch-a-sketch module.
- lab6: Interrupt Driven I/O.
- lab7: Serial RF-Communications.

These labs will expose you to a range of design activities, tool functionalities and implementation technologies. Each lab has a set of deliverables. These typically include the circuit schematics, demonstration of the hardware implementation and a complete report.

Due Dates

There will be seven labs throughout the term. The following are the due dates.

Item	Takes Place	Due Date	Topic
Lab #0	Fri (Week #2)	-	Equipment used in the Lab
Lab #1	Fri (Week #3)	Fri (Week #4)	Introduction to MC68HC12 Controller
Lab #2	Fri (Week #4)	Fri (Week #5)	Parallel Port Programming
Lab #3	Fri (Week #5)	Fri (Week #6)	Keypad & Switch Debouncing
Lab #4	Fri (Week #6)	Fri (Week #8)	Servo Motor Control/LCD Interface
Lab #5	Fri (Week #8)	Fri (Week #9)	High-Tech Etch-a-sketch
Lab #6	Fri (Week #9)	Fri (Week #10)	Interrupt Driven I/O
Lab #7	Fri (Week #10)	Fri (Week #11)	Serial RF-Communication

4 Demonstration & Writeup

Demonstration

Be prepared to:

- demonstrate the operation of the system
- explain how your design works
- explain how the components of the system works
- answer questions related to your design.

Writeup

1. **Problem Statement**.
   (a) Briefly summarize the problem solved in the lab.
2. **Assumptions and Constraints**
3. **System Overview & Justification of Design**
   (a) Give an overview of the system to be designed.
   (b) Briefly explain how the system works and reasons behind the design.
4. **Hardware**
   (a) brief explanation of the hardware.
   (b) for each component give a brief overview, pinout, pin description as required.
   (c) any required calculations for hardware components (i.e Resistor values, e.t.c).
   (d) complete wiring diagram for the system (multiple if necessary).
   (e) timing diagram if necessary.
5. **Software**
   (a) explain the function of your developed software, including subroutines.
   (b) software flowcharts as necessary.
   (c) any requierd calculations for software components (i.e., delay loops).
6. **System Manual**
   (a) give a brief user's manual for the system.
   (b) include information as required on how to start the program, required user input, how to interpret results.
7. **Error Analysis**
   (a) describe any problems with the system.
   (b) if no problems in the final system, describe problems/errors encountered during the development and how they were resolved.
8. **Performance Analysis**

(a) assess performance of the system (how optimized it currently is, how fast it is, e.t.c).

(b) suggest any desired improvements.

9. **Appendix**

(a) include all your code with comments.

(b) other relevant material as required.

5 Academic Misconduct

The policy for this course is zero tolerance for any form of academic misconduct. Consultation with other students is encouraged especially on design issues. However, directly copying another student's work or copying portions of code for example assembly language code) is an honor code violation and will result in a failing grade and may result in a failing grade in the course. Students will automatically be refered to the Director of the School and Dean of the college for action. **Please refer to the regulations outlined in the student handbook regarding academic misconduct.**

6 Appendix A – D-Bug12 Commands

An HCS12 demo board may have its micro-controller programmed with the D-Bug12 monitor to support debug activities. The D-Bug12 monitor occupies the flash memory and supports a set of debug commands. The user can use these commands to display and modify register and memory contents, set breakpoints at certain locations, step through the program, and download the program onto the demo board for execution. Please refer to Chapter 3, Section 3.2 "Development Tools for the HCS12" for detailed explanation of the commands listed below in the table.

Command	**Description**
ASM $< address >$	Single line assembler/disassembler
BAUD $< BAUDRate >$	Set the SCI communication baud rate
BF $< StartAddress >< EndAddress >$ [$< Data >$]	Block Fill user memory with data
BR [$< Address >< Address >$ ..]	Set/display user breakpoints
BULK	Bulk erase on-chip EEPROM
CALL [$< Address >$]	Execute a user subroutine; return to D-Bug12
G [$< Address >$]	**Go – begin execution of user program**
GT $< address >$	Go Till – set a temp breakpoint and begin execution of user program
HELP	Display D-Bug12 command set and command syntax
LOAD [$< AddressOffest >$]	Load user program in S-record format
MD $< StartAddress >$ [$< EndAddress >$]	Memory Display - display memory contents
MDW $< StartAddress >$ [$< EndAddress >$]	Memory Display Word - display memory contents
MM $< Address >$ [$< data >$]	Memory Modify - interactively change memory
MMW $< Address >$ [$< data >$]	Memory Modify Word - Change Memory Contents
MOVE $< StartAdd >< EndAdd >< DestAdd >$	Move a block of memory
NOBR [$< Address >< Address >$ ..]	Remove individual user breakpoint
RD	Register Display – display the CPU registers
RM	Register Modify – examine/change CPU Regs
T [$< Count >$]	Trace – execute an instruction, disassemble it and display the CPU registers
UPLOAD $< StartAddress >< EndAddress >$	Display memory contents in S-Record format
VERF [$< AddressOffset >$]	Verify memory contents against S-Record Data

Table 1: D-Bug12 Command Set Summary

7 Appendix B – D-Bug12 Commands Examples

Examples of using the D-Bug12 Commands are illustrated in your text book Section 3.5.3

MD "Memory Display"

The Memory Display command is used to display memory contents
syntax

```
MD <StartAddress>
```

The following example will display the contents of memory starting at 1000.

```
> MD 1000
  1000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
```

RM "Register Modify"

The Register Modify command is used to examine and/or modify the contents of the CPU12 registers interactively
syntax The following example will display the contents of registers

```
> RM
```

PC = 0000 **1500**
SP = 0A00
...
PC =1500
>

BF "Block Fill"

The Block Fill command is used to place a single 8-bit value into a range of memory locations.
syntax

```
BF <StartAddress> <EndAddress> [<Data>]
```

The following example will fill memory from location 4000 to 4100 with zeros.

```
> BF 4000 4100 0
```

CALL "Call Subroutine"

The CALL command is used to execute a subroutine and return to the D-Bug12 monitor program when the final RTS of the subroutine is executed. When control is returned to D-Bug12, the CPU register contents are displayed.
syntax

```
CALL  [<Address>]
```

The following example will illustrate the CALL operation.

```
> CALL 820
Subroutine Call Returned

PC      SP     X     Y     D = A:B      CCR= SXHI NZVC
0820   0A00   057C  0000     0F:F9           1001 0000
>
```

BR "Breakpoint Set"

The BR command is used to set a software breakpoint at a specified address or to display any previously set breakpoints. The function of a breakpoint is to halt user program execution when the program reaches the breakpoint address. When a breakpoint address is encountered, D-Bug12 disassembles the instruction at the breakpoint address, prints the CPU12's register contents, and waits for a D-Bug12 command to be entered by the user.
syntax

```
BR [<Address> <Address> ..]
```

The following example will illustrate the BR command.

```
> BR 35ec 2f80 c592
Breakpoints: 35EC 2F80 C592
> BR
Breakpoints: 35EC 2F80 C592
>
```

NOBR "Remove Breakpoints"

The NOBR command can be used to remove one or more previously entered breakpoints.
syntax

```
NOBR [<Address> <Address> ..]
```

The following example will illustrate the BR command.

```
> BR 800 810 820 830
Breakpoints: 0800 0810 0820 0830
> NOBR 810 820
Breakpoints: 0800 0830
>
```

T "Trace"

The Trace command is used to execute one or more user program instructions beginning at the current Program Counter (PC) location. As each program instruction is executed, the CPU12's register contents are displayed and the next instruction to be executed is displayed. The following example will illustrate the Trace command.

```
> T
PC     SP     X     Y          D = A:B     CCR  =  SXHI  NZVC
0803   09FE   057C  0000          10:00           1001  0000
0803    830001           SUBD   #$0001
```

8 Appendix C – Using D-Bug12 Routines

The D-Bug12 monitor provides a few subroutines to support I/O operations. One can utilize these I/O routines to facilitate program development. The D-Bug12 currently provides access to 18 different utility routines through a table of 16-bit pointers beginning at address $EE80 as shown in Table 2. This appendix will provide you with some information that will allow you to utilize internal D-Bug12 routines.

Please refer to Page 173, Section 4.8 in your text book "Using the D-Bug12 Functions to Perform I/O Operations for detailed examples.

Function	Description	Vector Table Address
far main()	Start of D-Bug12	$EE80
getchar()	Get a char from SCI0 or SCI1	$EE84
putchar()	send a char out SCI0 or SCI1	$EE86
printf()	Formatted Output - binary to char	$EE88
far GetCmdLine()	Obtain a line of input from the user	$EE8A
far sscanhex()	Convert and ASCII hex to binary int	$EE8E
isxdigit()	Checks for membership [0..0,a..f,A..F]	$EE92
toupper()	Converts lower case char to upper case	$EE94
isalpha()	Checks for membership in [a..z,A..Z]	$EE96
strlen()	Returns the length of a string	$EE98
strcpy()	Copies a null terminated string	$EE9A
far out2hex()	Displays 8-bit number as 2 ASCII chars	$EE9C
far out4hex()	Displays 16-bit number as 4 ASCII chars	$EEA0
SetUserVector()	Setup user interrupt service routine	$EEA4
far WriteEEByte()	Write a data byte to on-chip EEPROM	$EEA6
far EraseEE()	Bulk erase on-chip EEPROM	$EEAA
far ReadMem()	Read data from M68HC12 memory map	$EEAE
far WriteMem()	Write data to M68HC12 memory map	$EEB2

Table 2: D-Bug12 Utility Routines Summary

Here are a set of rules that you can follow to utilize these utility routines:

1. Calling a function from assembly language is simple.
   - First, **push the parameters onto the stack** in the proper order, loading the **first or only function parameter** into the D Accumulator. In other words if you have two parameters or arguments then one will be passed by the D Accumulator and the other pushed onto the stack.
   - Then call the function with a JSR instruction.
   - The code following the JSR instruction should remove any parameters pushed onto the stack. If a single parameter is stacked, a PULX or PULY instruction is one of the most efficient ways to unstack it.
2. All 8-bit and 16-bit function results are returned in the D Accumulator. Char values returned in the D accumulator are located in the 8-bit B Accumulator.
3. Boolean function results are zero for False and non-zero values for True.

Example #1:
To output a single character to the control terminal SCI (i.e., screen) you can use the **int putchar(int);** function whose "Pointer Address" is located at $EE86. If the control SCI's transmit data register is full when the function is called, **putchar()** will wait until the transmit data register is empty before sending the character.

Adding the following instruction sequence in your program will output the character **"A"** to the control SCI:

```
putchar         equ       $EE86
                ...
                ldd       #'A'
                jsr       [putchar,PCR]
                ...
```

The addressing mode used in the jsr instruction of this example is a form of **indexed indirect addressing** that uses the program counter as an index register. In reality, the HCS12 does not support PCR. Instead, the PCR mnemonic is used to instruct the assembler to calculate an offset to the address specified by the label **putchar**. The MiniIDE software supports this syntax. However, if you are using an assembler that does not support program-counter relative indexed addressing the following two instruction sequence can be used instead:

```
    ldx       putchar       ; load the address of putchar
    jsr       0,x           ; call the subroutine
```

Important: If the name of a library function is preceded by the keyword *far*, then it is located in the expanded memory and must be called by using the **call** instruction.

Example #2:
To retrieve a single character from the control terminal SCI use the **int getchar(void);** function whose pointer address is located in "$EE84". If an unread character is not available in the receive data register when this function is called, it will wait until one is received. Because the character is returned as an integer, the 8-bit character is placed in accumulator B.

Adding the following instruction sequence in your program will read a character from the SCI0 port

```
getchar         equ       $EE84
                ...
                jsr       [getchar,PCR]
                ...
```

Example #3:
The following is a sample code for utilizing the getchar() and printf() utility routines Notice that there are two org statements. The first org statement is to set the starting address of the variables and constants at location $1000. The second org statement is to set the start address of the main program to be executed to location $1100. When you attempt to run your program you need to issue the command g $1100 in the miniIDE environment.

```
CR          EQU   $0D                          ; ASCII Code for Carriage Return
LF          EQU   $0A                          ; ASCII Code for Line Feed
getchar     EQU   $EE84                        ; Address of Debug12 Utility Function getchar()
printf      EQU   $EE88                        ; Address of Debug12 Utility Function printf()

            ORG   $1000                        ; Start Address of variables/constants
```

```
msg         db    "char pressed is %c",CR,LF,0

            ORG   $1100                       ; Start Address of main program
getnewval   LDX   getchar                     ; get the character from the keyboard
            JSR   0,X

            CMPB  #$1B                        ; Check if the char is an ESC
            BEQ   terminate                   ; jump to end of program

            PSHD                              ; push char on stack (parameter for printf)
            LDD   #msg                        ; second parameter in ACC D
            LDX   printf                      ; Load Index Reg X with printf address
            JSR   0,X                         ; print message on terminal

            BRA   getnewval                   ; repeat getting new chars
terminate   SWI                               ; end of program (return to D-Bug12)
```

9 Appendix D – Evaluation Board "Memory Usage"

The EVB's memory usage and requirements are described below and summarized in Table 3 and Figure 3 respectively. **Please refer to Page 94 Section 3.4 in the textbook for further information on the D-Bug12 Memory Map.**

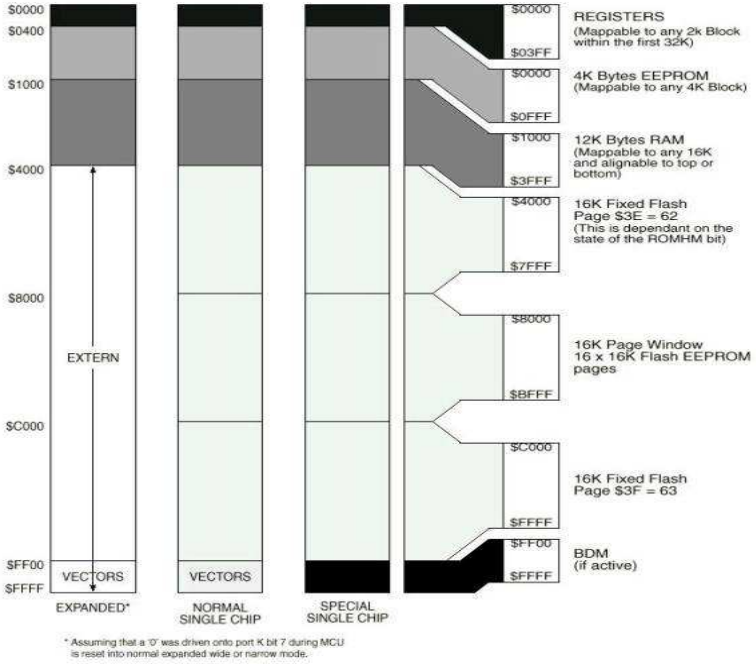

Figure 3: The MC9S12DG256 Memory Map

Address Range	Description	Location
$0000 - $03FF	CPU I/O Registers	on-chip (MCU)
$0400 - $0FFF	On-chip EEPROM	on-chip (MCU)
$1000 - $3BFF	On-chip SRAM (For User)	on-chip (MCU)
$3C00 - $3FFF	On-chip SRAM (D-Bug12)	on-chip (MCU)
$4000 - $EE7F	D-Bug12 Code	on-chip Flash (MCU)
$EE80 - $EEBF	User Accessible func Table	Fixed Flash
$EEC0 - $EEFF	D-Bug12 customization data	
$EF00 - $EF8B	D-Bug12 startup code	
$EF8C - $EFFF	rest/interrupt table	
$F000 - $FFFF	Bootloader	

Table 3: Memory Map for M68HC12A4EVB

1. **Important:** The user has 11 KB of SRAM available for development of application programs. The user code can start at address $1000 upto $3BFF
2. Part of the on-chip EEPROM (4K) is available for user programs. Even though the HCS12DX256 parts contain 4 KB of EEPROM, only the upper 3 KB are visible, as the lower 1 KB is overlaid with the I/O registers.

3. The monitor program, D-Bug12, occupies locations $4000 upto $EE7F.

Below is an example of a program that starts at memory location $1000 (On-chip RAM).

```
ORG    $1000    ; Starting Address of Program
ldaa   $1500    ; A <== [$1500]
adda   $1502    ; A <== [A] + [$1502]
suba   $1505    ; A <== [A] - [$1505]
staa   $1510    ; $1510 <== [A]
end
```

10 Appendix E – MCU Connections

Eight pin header connectors (H1 .. H8), surrounding the breadboard provide access to the MCU's I/O and bus lines as seen in Figure 4. These connectors are located adjacent to the prototype area.

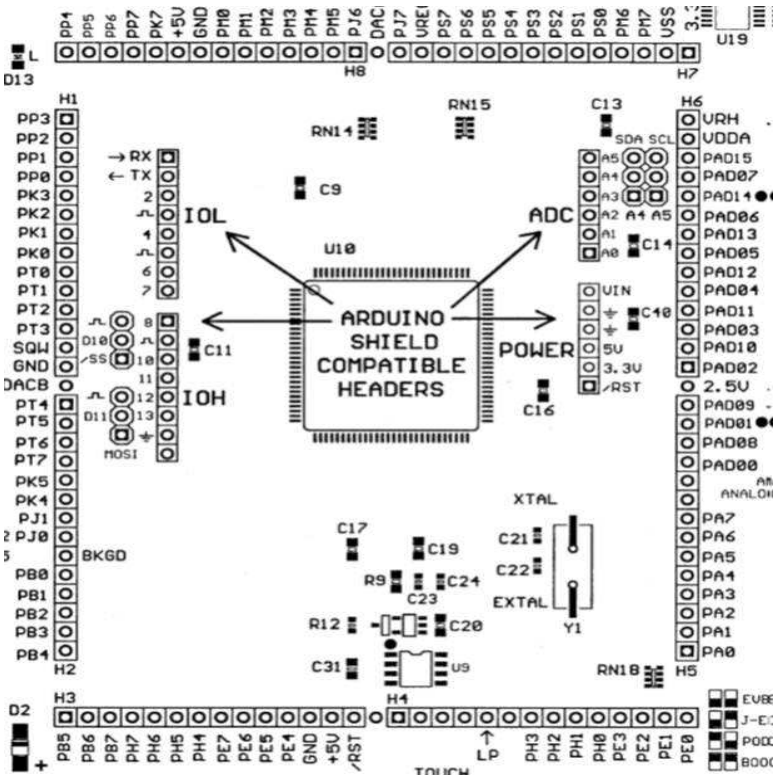

Figure 4: Main Pin Headers on the Dragon12-Plus2 Board

1. Connector H1 located at upper/left of breadboard.
2. Connector H2 located at lower/left of breadboard (below H1).
3. Connector H3 located at bottom/left of breadboard.
4. Connector H4 located at bottom/right of breadboard.
5. Connector H5 located at lower/right of breadboard.
6. Connector H6 located at upper/right of breadboard.
7. Connector H7 located at top/right of breadboard.
8. Connector H8 located at top/left of breadboard.

Table 4 provides a brief description of the signals.

I/O Pin Usage

Many I/O pins of the MC9S12DG256 on the Dragon12-Plus2 board are used by on-board peripherals, here are examples:

1. PORT A: Keypad.

Connector	Signal Name	Port	Description
H1	PP3, PP2, PP1, PP0	Port P(0..3)	Gen Purpose I/O
H1	PK3, PK2, PK1, PK0	Port K(0..3)	Gen Purpose I/O
H1	PT0, PT1, PT2, PT3	Port T(0..3)	NOT USED
H2	PT4, PT5, PT6, PT7	Port T(4..7)	NOT USED
H2	PK5, PK4	Port K(4..5)	Gen Purpose I/O
H2	PJ1, PJ0	Port J(0..1)	NOT USED
H2	PB0, PB1, PB2, PB3, PB4	Port B(0..4)	Gen Purpose I/O
H3	PB5, PB6, PB7	Port B(5..7)	Gen Purpose I/O
H3	PH7, PH6, PH5, PH4	Port H(4..7)	Gen Purpose I/O
H3	PE7, PE6, PE5, PE4	Port E(4..7)	NOT USED
H4	PH3, PH2, PH1, PH0	Port H(0..3)	Gen Purpose I/O
H4	PE3, PE2, PE1, PE0	Port E(0..3)	NOT USED
H5	PA0, PA1, PA2, PA3, PA4, PA5, PA6, PA7	Port A(0..7)	Gen Purpose I/O
H5	PAD00, PAD08, PAD01, PAD09	Port AD0(0,1,8,9)	Analog-to-Dig
H6	PAD02, PAD03, PAD04, PAD05, PAD06, PAD07	Port AD0(2..7)	Analog-to-Dig
H6	PAD10, PAD11, PAD12, PAD13, PAD14, PAD15	Port AD1(0..5)	Analog-to-Dig
H7	PS0, PS1, PS2, PS3, PS4, PS5, PS6, PS7	Port S(0..7)	Serial Comm
H7	PM6, PM7	Port M(6,7)	NOT USED
H8	PM0, PM1, PM2, PM3, PM4, PM5	Port M(0..5)	NOT USED
H8	PP4, PP5, PP6, PP7	Port P(4..7)	Gen Purpose I/O

Table 4: MCU Connectors J8/J9 Pin Assignments

2. PORT B: Light Emmitting Diodes (LEDs)
3. PORT H: DIP and Pushbutton Switches
4. PORT K: Liquid Crystal Display (LCD)
5. PORT P: 7-Segment Display
6. PORT AD: Light Sensor, Temp Sensor, Trimmer Pot

IMPORTANT:
Fortunetly, it's unlikely that all on-board peripherals will be used by one application program. So the I/O pins on unused peripheral devices can still be used by your circuits on the breadboard. For instance, if you don't touch the 4x4 on-board keypad, the entire port A will be available to your circuits.

11 Appendix F – Port Register

Each port within the M68HC12 has several registers that are used to control its functionality and also give the status of the port following a read/write transaction. Table 5 provides a brief description of the regular I/O ports and their associated registers. Ports A, B, E, K can be used for Input/Output based on how the "Data Direction" register has been programmed.

Port	Register	**Description**	**Address**	**Current Usage**
PORT A	PORTA	Data Register **A**	$0000	Keypad
	DDRA	Data Direction Register **A**	$0002	
PORT B	PORTB	Data Register **B**	$0001	LED
	DDRB	Data Direction Register **B**	$0003	
PORT P	PORTP	Data Register **P**	$0258	7-Seg Display
	DDRP	Data Direction Register **P**	$025A	
PORT H	PORTH	Data Register **H**	$0260	DIP Switches
	DDRH	Data Direction Register **H**	$0262	
PORT E	PORTE	Data Register **E**	$0008	IRQ, XIRQ
	DDRE	Data Direction Register **E**	$0009	
PORT K	PORTK	Data Register **K**	$0032	LCD
	DDRK	Data Direction Register **K**	$0033	

Table 5: M68HC12 I/O Ports and Associated Registers

Table 6 provides a brief description of the I/O ports (H,J) that have interrupt capability (Wakeup). Port P is very similar to ports H and J in functionality.

Port	Register	**Description**	**Address**
PORT H	PTH	Data Register **H**	$0260
	DDRH	Data Direction Register **H**	$0262
	PTIH	Input Register **H**	$0261
	RDRH	Reduced Drive Register **H**	$0263
	PERH	Pull Devide Enable **H**	$0264
	PPSH	Polarity Select Register **H**	$0265
	PIEH	Interrupt Enable Register (Wakeup)	$0266
	PIFH	Interrupt Flag Register **H**	$0267
PORT J	PTJ	Data Register **J**	$0268
	DDRJ	Data Direction Register **J**	$026a
	PTIJ	Input Register **J**	$0269
	RDRJ	Reduced Drive Register **J**	$026b
	PERJ	Pull Device Enable **J**	$026c
	PPSJ	Polarity Select Register **J**	$026d
	PIEJ	Interrupt Enable Register **J**	$026e
	PIFJ	Interrupt Flag Register **J**	$026f

Table 6: I/O Ports with Interrupt Enable Capability

Table 7 provides a brief description of other I/O ports and registers that are used for "Real Time Interrupt" (Section 6.7 in Text Book), "Analog-to-Digital Conversion" (Section 12.3 in Text Book), and "Serial Communication".

Port	Register	**Description**	**Address**
RTI	RTICTL	Real Time Interrupt Control Register (RTR0:RTR6)	$003B
	CRGFLG	The CRG Flag Register (RTIF)	$0037
	CRGINT	The CRG Interrupt Enable Register (RTIE)	$0038
PORT ATD	ATD0CTL0	Control Register 0 (Reserved)	$0080
	ATD0CTL1	Control Register 1 (Reserved)	$0081
	ATD0CTL2	Control Register 2 (ADPU)	$0082
	ATD0CTL3	Control Register 3 (S8C, S4C, S2C, S1C)	$0083
	ATD0CTL4	Control Register 4 (PRS4:PRS0)	$0084
	ATD0CTL5	Control Register 5 (SCAN, MULT, CC, CB, CA)	$0085
	ATDSTAT0	Status Register 0 (SCF)	$0086
	ATDSTAT1	Status Register 1 (Reserved)	$0087
	PORTAD0	Data Input (Bit7:Bit0)	$008F
	ATD0DR0H	Result Register 0 (High Byte)	$0090
	ATD0DR7H	Result Register 7 (High Byte)	$009E
PORT SCI 0	SCI0BDH	Baud Rate Register High	$00C8
	SCI0BDL	Baud Rate Register Low	$00C9
	SCI0CR1	Control Register 1	$00CA
	SCI0CR2	Control Register 2	$00CB
	SCI0SR1	Status Register 1	$00CC
	SCI0SR2	Status Register 2	$00CD
	SCI0DRH	Data Register High	$00CE
	SCI0DRL	Data Register Low	$00CF

Table 7: M68HC12 Ports and Registers

12 Appendix G – Dragon12-Plus2 Displays

The Dragon12-Plus2 Board has several means to display information by the MCU12. Here are some examples:

1. Light Emmitting Diodes, (LEDs) (connected to Port B)
2. Seven-Segment Displays, (connected to Port P)
3. Liquid Crystal Display, (connected to Port K).

Each peripheral is connected to a specific port as demonstrated above. These ports can still be used as general purpose ports if the peripherals are not used.

1. If you don't use the LCD or just unplug the LCD, the **Port K** will be available as well.
2. **Port B** drives LEDs, but if you ignore the status of the LEDs, the port B can drive any other I/O devices on the breadboard.

Please refer to the "Dragon12-Plus2 Trainer, User's Manual" for more details.

LEDs:

Each port B line is monitored by an LED. In order to turn on port B LEDs, the PJ1 (port J pin 1) must be programmed as output and set for logic zero.

7-Segment LED multiplexing

There are 4 digits of 7-Segment LEDs on the Dragon12-Plus2 board. The type of the 7-Segment display on board is called common cathode. in an individual digit, all anodes are driven individually by an output pot and all cathodes are internally connected together. Before sending a number to a 7-Segment LED, the number must be converted to its corresponding 7-segment code depending how the 7-segment display is connected to an output port.

The Dragon12-Plus2 board uses port B to drive 7-segment anodes and used PP0-PP3 (i.e., Port P) to drive common cathodes. In the next few paragraphs we will explain how to multiplex 7-segment by displaying the number 1234 on the display. By convention, the 7-Segments are called segment a, b, c, d, e, f and g, as shown in Figure 5.

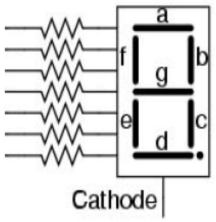

Figure 5: The Dragon12 7-Segment

The segment a, b, c, d, e, f, g and Decimal Point are driven by (PORT B) PB0, PB1, PB2, PB3, PB4, PB5, PB6, PB7, respectively. The hex value of the segment code is shown in the following table:

The schematic for multiplexing 4 digits is shown in Figure 7.

Number	DP	G	F	E	D	C	B	A	Hex Value
1	0	0	0	0	0	1	1	0	$06
2	0	1	0	1	1	0	1	1	$5B
3	0	1	0	0	1	1	1	1	$4F
4	0	1	1	0	0	1	1	0	$66

Figure 6: Hex Value of the numbers '1, 2,3, 4'

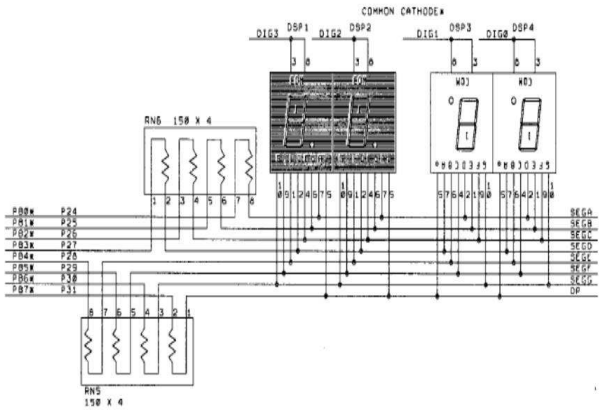

Figure 7: Multiplexing the 7-Segment Displays on the Dragon12-Plus2 Board

The digits **3, 2, 1 and 0** are driven by PP0, PP1, PP2 and PP3, respectively (PORT P). The 7-Segment LED is turned on one at a time at 250 HZ refresh rate. It's so fast that our eyes will perceive that all digits are turned on at the same time. To display the number '1234' on the 7-Segment display, the following steps should be taken:

1. Output $06 to Port B, set PP0 low and PP1, PP2 and PP3 high. The number 1 is shown on the digit 3 (the leftmost digit) but other 3 digits are turned off.
2. Delay 1ms.
3. Output $5B to Port B, set PP1 low and PP0, PP2 and PP3 high. The number 2 is shown on the digit 2 but other 3 digits are turned off.
4. Delay 1ms.
5. Output $4F to Port B, set PP2 low and PP0, PP1 and PP3 high. The number 3 is shown on the digit 1 but other 3 digits are turned off.
6. Delay 1ms.
7. Output $66 to Port B, set PP3 low and PP0, PP1 and PP2 high. The number 4 is shown on the digit 0 (the rightmost digit) but other 3 digits are turned off.
8. Delay 1ms.
9. Go back to step 1.

13 Appendix H – Dragon12-Plus2 Keypad and DIP Switches

The Dragon12-Plus2 Board has several means to send information to the MCU12. Here are some examples:

1. Hex Key Pad, (connected to Port A)
2. DIP Switches, (connected to Port H)
3. Push Buttons, (connected to Port H)

Each peripheral is connected to a specific port as demonstrated above. These ports can still be used as general purpose ports if the peripherals are not used. Please refer to the "Dragon12-Plus2 Trainer, User's Manual" for more details.

DIP Switch and Pushbuttons:

Port H is connected to an 8-position DIP switch. The DIP switch is connected to VCC via the RN4 (four 2.7K resistors) and RN5 (four 2.7K resistors). When Port H is programmed as an output port, the DIP switch setting is ignored.

Keypad:

Port A is an 8-bit bi-directional port. Its primary usage is for a 4x4 keypad. If the port is not used for the keypad, it can be used as a general purpose I/O. The Dragon12-Plus2 board uses port A to scan and test if a key has been pressed. The schematic for the keypad connection is shown in Figure 8.

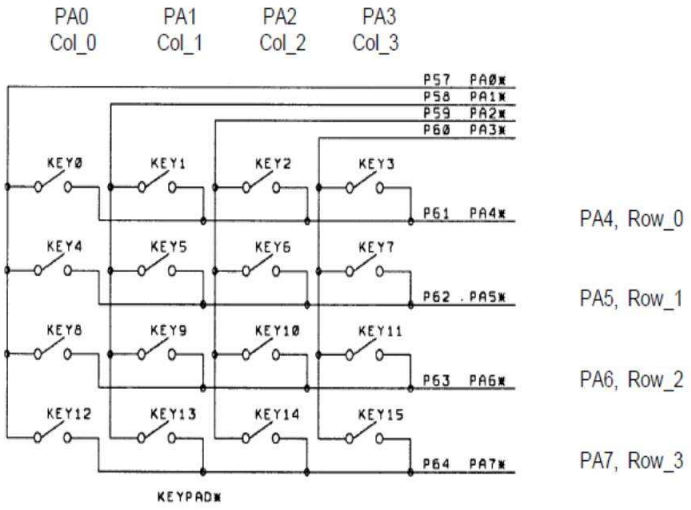

Figure 8: The Dragon12 Keypad

Keypad Connections:

- PA0 connects COL0 of the keypad.
- PA1 connects COL1 of the keypad.

- PA2 connects COL2 of the keypad.
- PA3 connects COL3 of the keypad.
- PA4 connects ROW0 of the keypad.
- PA5 connects ROW1 of the keypad.
- PA6 connects ROW2 of the keypad.
- PA7 connects ROW3 of the keypad.

The following explains how to scan and test if a specific key has been pressed on the keypad.

1. Keypad scan routine sets PA3 low and PA0, PA1, PA2 high, then tests PA4-PA7.
    - If no key is pressed, PA4-PA7 remain high.
    - If PA7 = low, the key 15 (**D**) is pressed.
    - if PA6 = low, the key 11 (**C**) is pressed.
    - If PA5 = low, the key 7 (**B**) is pressed.
    - If PA4 = low, the key 3 (**A**) is pressed.
2. Keypad scan routine sets PA2 low and PA0, PA1, PA3 high, then tests PA4-PA7.
    - If no key is pressed, PA4-PA7 remain high.
    - If PA7 = low, the key 14 (#) is pressed.
    - if PA6 = low, the key 10 (**9**) is pressed.
    - If PA5 = low, the key 6 (**6**) is pressed.
    - If PA4 = low, the key 2 (**3**) is pressed.
3. Keypad scan routine sets PA1 low and PA0, PA2, PA2 high, then tests PA4-PA7.
    - If no key is pressed, PA4-PA7 remain high.
    - If PA7 = low, the key 13 (**0**) is pressed.
    - if PA6 = low, the key 9 (**8**) is pressed.
    - If PA5 = low, the key 5 (**5**) is pressed.
    - If PA4 = low, the key 1 (**2**) is pressed.
4. Keypad scan routine sets PA0 low and PA1, PA2, PA3 high, then tests PA4-PA7.
    - If no key is pressed, PA4-PA7 remain high.
    - If PA7 = low, the key 12 (*) is pressed.
    - if PA6 = low, the key 8 (**7**) is pressed.
    - If PA5 = low, the key 4 (**4**) is pressed.
    - If PA4 = low, the key 0 (**1**) is pressed.

14 Appendix I – S-Record Format

The S-Record format for output modules was devised for the purpose of encoding programs or data files in a printable format for transportation between computer systems.

14.1 S-Record Content

S-Records are essentially character strings made of several fields that identify the recored type, record length, memory address, code/data, and checksum. Each byte of binary data is encoded as a 2-character hexadecimal number. The 5 fields that comprise an S-record are shown below
The S-Record fields are composed as follows:

Type	Record Length	Address	Code/Data	CheckSum

1. **Type Field**: 2 characters, *Contents*: S-Record types, S0, S1, e.t.c
2. **Record length Field**: 2 characters, Contents: the count of the character pairs in the record
3. **Address Field**: 4,6, or 8 characters, *Contents*: the 2-,3-,or 4-byte address at which the data field is to be loaded into memory.
4. **Code/data Field**: 0-2n characters, *Contents*: from 0 to n bytes of executable code, memory-loadable data, or descriptive information.
5. **Checksum**: 2 characters, *Contents*: The least significant byte of the 1's complement of the sum of the values represented by the pairs of characters making up the record length, address, and the code/data fields.

14.2 S-Record Types

Eight types of S-records have been defined to accomodate the several needs of the encoding, transportation, and decoding functions. Table 8 describes the different S-records types.

Type	Description
S0	The header record for each block os S-records.
S1	A record containing code/data
S9	A termination record for a block of S1 records.

Table 8: S-Record Types

14.3 S-Record Example

Shown below is a typical S-record format module, as printed or displayed:

```
S0030000FC                  --> 03 indicates 3 char pairs follow.
S10B08008604C605180620FE5B
S9030000FC
```

The above module consists of an S0 header record, one S1 code/data record, and an S9 termination record.

15 Appendix J – Using Interrupts with D-Bug12

An interrupt is a hardware solution for both even detection and changing the program flow to a service routine. When using an interrupt, the service routine is called the interrupt service routine or ISR.

15.1 Interrupt Program Flow

Figure 9 shows the overall program flow in an interrupt driven system. The main program, or background, is shown running an endless task loop. Once an interrupt event is detected, the background will be interrupted and the CPU will automatically jump to and execute the interrupt service routine. This is like a subroutine call instruction that is initiated by a hardware event. The interrupt service routine ends with an *"rti"* instruction, which returns the CPU to the background to resume execution at the instruction that would have run if the interrupt had not occurred.

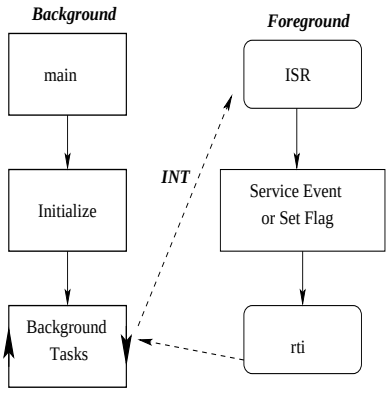

Figure 9: Interrupt Program Flow

15.2 68HC12 Interrupt Sources and Setup

We can divide the interrupt sources into four groups: interrupts from on-chip resources (Timers, A/D, ..), external interrupts (IRQ,XIRQ, ..) software interrupts (SWI, ..) and reset exceptions. An interrupt vector is two fixed memory locations that contain the address of the interrupt service routine for a given interrupt. The CPU12 interrupt vectors are stored at $FF80-$FFFF. Table 9 shows the interrupt sources and vector location for some interrupt sources on the M68HC12.

Interrupt Source	Vector Address	CCR Mask	Local Enables
Analog to Digital Conv	$FFD2:$FFD3	I	ATD0CTL2(ASCIE)
SCI 1 Serial System	$FFD4:$FFD5	I	SC1CR2(TIE, TCIE, RIE, ILIE)
SCI 0 Serial System	$FFD6:$FFD7	I	SC0CR2(TIE, TCIE, RIE, ILIE)
SPI Serial System	$FFD8:$FFD9	I	SP0CR1(SPIE,SPTIE)
Timer Channel 7	$FFE0:$FFE1	I	TIE(C7I)
Timer Channel 0	$FFEE:$FFEF	I	TIE(C0I)
Real Time Interrupt	$FFF0:$FFF1	I	CRGINT(RTIE)
IRQ	$FFF2:$FFF3	I	IRQCR(IRQEN)

Table 9: 68HC12 Interrupt Sources & Vector Addresses

So, for example, to load the interrupt vector for "Timer Channel 0", we would use the following code: This loads the address of the Ic0Isr interrupt service routine into $FFEE:$FFEF, which is the vector for Timer Channel 0. Now when a rising edge is detected on PORTT, bit 0, the CPU saves the context, sets the I mask, and loads the program counter with the contents of address $FFEE:$FFEF, which is the address of the ISR.

```
            org   $FFEE
            FDB   Ic0Isr      ; Ic0Isr is the location where the ISR is located
            ....
            ....
Ic0Isr      movb #C0F, TFLG1  ; Ic0Isr is the ISR for Timer Channel 0
            ldd  PulseCnt     ; increment pulse count
            addd #1
            std  PulseCnt
            rti
```

Figure 10 illustrates the concept graphically.

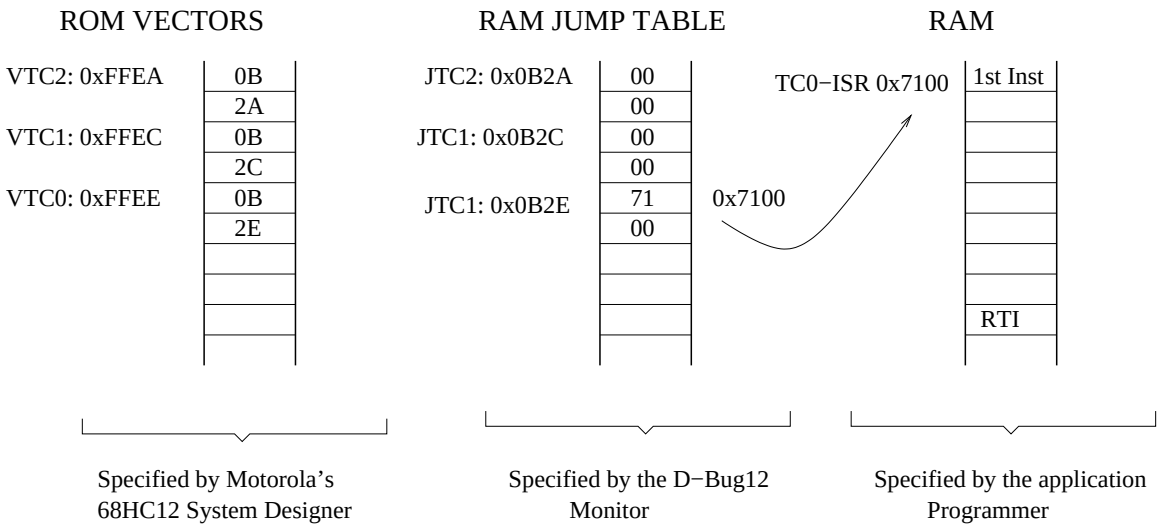

Figure 10: Memory Map of the Interrupt Vector Linkage for TC2

15.3 Using Interrupts with D-Bug12

When we are writing programs that use interrupts under the D-bug12 monitor, it is impractical to change the actual interrupt vectors. The vectors are stored in ROM with D-Bug12 and some vectors are used by D-Bug12. Even if they are in Flash ROM, it is not practical to reprogram the whole Flash ROM every time we write or change a program.

To accommodate interrupts, D-Bug12 includes a utility routine, *SetUserVec()*, that will load the ISR address into a jump table located in monitor RAM. The actual vector contains the address of an interrupt dispatch program in D-Bug12. If there is a user ISR address in the jump table, the dispatch program jumps to that address. If no user ISR address exists in the jump table, the dispatch program displays a message indicating that an unexpected interrupt occurred.

To use *SetUserVec()*, we pass the ISR address and a vector number that corresponds to the interrupt source. The vector numbers for *SetUserVec()* are shown in Table 10 (**Please check Page 273 Table 6.3 in your text book for details**). For example, if we need to use the Timer Channel 0 interrupt, the vector number would be 55.

Interrupt Source	Ram Vec Address	Vec Number
PORTH Key Wake-up	$3E4C	38
PORTJ Key Wake-up	$3E4E	39
A-to-D1	$3E50	40
A-to-D0	$3E52	41
SCI#1	$3E54	42
SCI#0	$3E56	43
SPI#0	$3E58	44
Pulse Accum. Edge	$3E5A	45
Pulse Accum. Overflow	$3E5C	46
Timer Overflow	$3E5E	47
Timer Channel 7 .. Timer Channel 0	$3E60 ..$3E6E	48 .. 55
RTI	$3E70	56
IRQ	$3E72	57
XIRQ	$3E74	58
SWI	$3E76	59
Unimplemented Opcode Trap	$3E78	60

Table 10: SetUserVec() Interrupt Vector Numbers

EXAMPLE #1:
It is very simple to set up the SRAM interrupt vector. Assuming that the service routine for the $\overline{IRQ}$ interrupt starts with the label **irqISR**. Recall that the RAM Vector Address of the IRQ is $3E72. Then one of the following three methods can be used to establish the SRAM table entry for the $\overline{IRQ}$ inteterrupt:

Method #1

```
org     $3E72
dc.w    irqISR
```

Method #2

```
ldd     #irqISR
std     $3E72
```

Method #3

```
setuservector   equ     $EEA4           ; address of Debug-12 utility function
                ldd     #irqISR         ; push irqISR address on stack
                pshd                    ;
                ldab    #57             ; load vector number of IRQ
                clra                    ;
                ldx     setuservector   ;
                jsr     0,x
                puld                    ; clean up the stack
```

EXAMPLE #2:
Assume that the $\overline{IRQ}$ pin of the HCS12DP256 is connected to a 1-Hz digital waveform and Port B is connected to eight LEDs. Write a program to configure Port B for output and enable the $\overline{IRQ}$ interrupt, and also write the service routine for the $\overline{IRQ}$ interrupt. The service routine for the $\overline{IRQ}$ interrupt simply increments a counter and outputs it to Port B.

Solution #1

```
UserIRQ equ     $3E72
DDRB    equ     3
DDRJ    equ     $26a
PTJ     equ     $268
PTB     equ     1
IRQCR   equ     $1e
        org     $1000
count   ds.b    1               ; reserve 1 byte for count
        org     $1500
        lds     #$1500          ; set up the stack pointer
        movw    #IRQISR,UserIRQ ; set up interrupt vector in SRAM
        clr     count
        movb    #$FF,DDRB       ; configure Port B for output
        bset    DDRJ,$02        ; configure PJ1 pin for output (required in Dragon12)
        bclr    PTJ,$02         ; enable LEDs to light (required in Dragon12)
        movb    count,PTB       ; display the count value on LEDs
        movb    #$C0,IRQCR      ; enable IRQ pin interrupt and select edge-triggering
        cli                     ;  "
forever nop
        bra     forever         ; wait for IRQ pin interrupt
; ---------------------------------------------------------------
; This is the IRQ service routine.
; ---------------------------------------------------------------
IRQISR  inc     count           ; increment count
        movb    count,PTB       ; and display count on LEDs
        rti
        end
```

Solution #2

```
setuservector   equ  $3E72
DDRB            equ     3
DDRJ            equ     $26a
PTJ             equ     $268
PTB             equ     1
IRQCR           equ     $1e
                org     $1000
count           ds.b    1          ; reserve 1 byte for count
                org     $1500
                lds     #$1500    ; set up the stack pointer
                ldd     #IRQISR    ; push IRQISR address on stack
                pshd               ;
```

```
                ldab    #57         ; load vector number of IRQ
                clra                ;
                ldx setuservector   ; Start calling setuservector
                jsr     0,x         ;
                puld                ; clean up the stack
                clr     count
                movb    #$FF,DDRB   ; configure Port B for output
                bset    DDRJ,$02    ; configure PJ1 pin for output (required in Dragon12)
                bclr    PTJ,$02     ; enable LEDs to light (required in Dragon12)
                movb    count,PTB   ; display the count value on LEDs
                movb    #$C0,IRQCR ; enable IRQ pin interrupt and select edge-triggering
                cli                 ; "
forever         nop
                bra     forever     ; wait for IRQ pin interrupt
; ---------------------------------------------------------------------------------------------
; This is the IRQ service routine.
; ---------------------------------------------------------------------------------------------
IRQISR          inc count     ; increment count
        movb count,PTB  ; and display count on LEDs
        rti
        end
```

EXAMPLE #3:
The following is yet another example of implementing an ISR with D-Bug12.

```
*************************************************************************
** Example: a program that uses the output compare function of timer
**          channel 7 (TC7) to create time delays.
** Assumptions: E-clock = 8MHz
** Functionality: The program performs 40 output compare operations on
**                the TC7 with each operation creating 25 ms delay.
**                Forty such operations will create a delay of one second
*************************************************************************
CR              equ     $0D             ; ASCII code of carriage return
LF              equ     $0A             ; ASCII code of line feed
tios            equ     $40             ; timer IC/OC select register
tcnt            equ     $44             ; timer count register
tscr1           equ     $46             ; timer system control register
tctl1           equ     $48             ; timer control register 1
TIE             equ     $4C             ; timer mask register 1
TSCR2           equ     $4D             ; timer mask register 2
tflg1           equ     $4E             ; timer flag register 1
tc7             equ     $5E             ; timer output compare register 7
pactl           equ     $60             ; pulse accumulator control register

printf          equ     $EE88           ; memory location to hold the
                                        ; starting address of printf()
setuservector   equ     $EEA4           ; memory location to hold the
                                        ; starting address of SetUserVector()
oc7_int_vect    equ     $FFE0
oc7_vec_num     equ     48

                org     $1000           ; start address of variables
oc7_cnt         rmb     1               ; number of output compare ops
                                        ; remained to be performed
seconds         rmb     1               ; number of seconds passed
                                        ; since the beginning
total           rmb     1               ; number of seconds remained

                org     $1100           ; starting address of the program
                lds     #$2000          ; initialize the stack pointer
                clr     seconds         ; start from 0 seconds
                jsr     init_oc7        ; initialize oc7 parameters
                ldaa    #10             ; store 10 in total
                staa    total

; the following 6 instructions set up oc7 interrupt vector
                ldd     #oc7_isr
                pshd                    ; push oc7_isr address on stack
                ldab    #oc7_vec_num    ; load vector number of oc7
                clra
                ldx     setuservector   ; setup D-bug12 for ISR
```

```
                jsr     0,x
                puld                    ; clean up the stack

; the following two instructions enable oc7 interrupt
                bset    TIE,$80       ; enable oc7 interrupt
                cli                     ; enable interrupt globally

; the outer loop make sure 10 second delays are created
outer_loop      tst     total
                beq     done
                movb    #40,oc7_cnt     ; set up the number of oc7 compares
                                        ;  to be performed
                ldd     #25000
                std     tc7             ; initialize tc7 to 25000 so that every
                                        ; creates 25 ms delay
; ********************************************************************
; the inner loop performs 40 oc7 output compare operations
; to create 1 second delay with each operation creates
; 25 milliseconds delay
; ********************************************************************
inner_loop      tst     oc7_cnt         ; wait for 40 oc7 operations to
                                        ; be performed
                bne     inner_loop      ; and oc7 interrupt will occur here
                inc     seconds         ; one second delay has been created
                clra
                ldab    seconds
                pshd
                ldd     #time_msg
                ldx     printf
                jsr     0,x             ; print out the message
                leas    2,sp
                dec     total
                jmp     outer_loop
done            swi

; ********************************************************************
; the following subroutine initializes the oc7 parameters
; ********************************************************************
init_oc7        movb    #$80,tscr1      ; enable timer, timer runs during wait state,
                                        ; and while in background mode,
                                        ; also clear flags normally
                clr     TIE             ; disable all output compare interrupts
                movb    #$80,tios       ; select channel 7 to act as
                                        ; output compare
                movb    #$2B,TSCR2      ; set the timer counter prescale factor
                                        ; to 8 and reset tcnt on a successful
                                        ; output compare 7 event
                movb    #$00,pactl      ; choose timer prescaler clock as
                                        ; timer counter clock
```

```
                rts

; ***************************************************************
; oc7 interrupt service routine
; ***************************************************************
oc7_isr         bclr    tflg1,$7F       ; clear c7F flag
                dec     oc7_cnt
                rti

time_msg        db      " %d seconds have passed",CR,LF,0
                end
```